

# Arquitectura de Computadores (AC)

## Cuaderno de prácticas.

### Bloque Práctico 2. Programación paralela II: Cláusulas OpenMP

Estudiante (nombre y apellidos): Alberto Jesús Durán López

Grupo de prácticas: 1

Fecha de entrega:

Fecha evaluación en clase:

#### Ejercicios basados en los ejemplos del seminario práctico

1. ¿Qué ocurre si en el ejemplo del seminario `shared-clause.c` se añade a la directiva `parallel` la cláusula `default(none)`? (añada una captura de pantalla que muestre lo que ocurre) **(b)** Resuelva el problema generado sin eliminar `default(none)`. Añada el código con la modificación al cuaderno de prácticas.

**RESPUESTA:** Al declarar “n” como constante, no existe problema alguno al compartir variables puesto que las variables constantes no tienen problemas a la hora de sincronizarse entre hebras.

**CÓDIGO FUENTE:** `shared-clauseModificado.c`

```
#include <stdio.h>
#ifdef _OPENMP
#include <omp.h>
#endif

int main()
{
    int i, n = 7;
    int a[n];
    for (i=0; i<n; i++)
        a[i] = i+1;

    #pragma omp parallel for shared(a,n) default(none)
    for (i=0; i<n; i++)
        a[i] += i;

    printf("Después de parallel for:\n");

    for (i=0; i<n; i++)
        printf("a[%d] = %d\n", i, a[i]);
}
```

#### CAPTURAS DE PANTALLA:

```
albduranlopez@albduranlopez:~/Escritorio/Tercera Entrega AC/codigo$ gcc -O2 -fopenmp shared-clause.c -o shared-clause
shared-clause.c: In function 'main':
shared-clause.c:13:11: error: 'n' not specified in enclosing parallel
    #pragma omp parallel for shared(a) default(none)
    ^
shared-clause.c:13:11: error: enclosing parallel
```

2. ¿Qué ocurre si en `private-clause.c` se inicializa la variable `suma` fuera de la construcción `parallel` en lugar de dentro? (inicialice `suma` a un valor distinto de 0 dentro y fuera de `parallel`) Razone su respuesta. Añada el código con la modificación al cuaderno de prácticas.

**RESPUESTA:** Si inicializamos la variable suma fuera del parallel contendría basura ya que private lo que hace es crear una variable privada llamada suma para cada hebra. Si se inicializa dentro de parallel no almacenará basura

**CÓDIGO FUENTE:** private-clauseModificado.c

```
#include <stdio.h>
#ifdef _OPENMP
#include <omp.h>
#else
#define omp_get_thread_num() 0
#endif
int main()
{
    int i, n = 7;
    int a[n], suma;

    for (i=0; i<n; i++)
        a[i] = i;

    suma=1;
    #pragma omp parallel private(suma)
    {
        //suma=0;
        #pragma omp for
        for (i=0; i<n; i++)
        {
            suma = suma + a[i];
            printf("thread %d suma a[%d] / ", omp_get_thread_num(), i);
        }

        printf("\n* thread %d suma= %d", omp_get_thread_num(), suma);
    }
    printf("\n");
}
```

**CAPTURAS DE PANTALLA:**

```
albduranlopez@albduranlopez:~/Escritorio/Tercera Entrega AC/codigo$ gcc -O2 -fopenmp private-clause.c -o dentro
albduranlopez@albduranlopez:~/Escritorio/Tercera Entrega AC/codigo$ ./dentro
thread 6 suma a[6] / thread 3 suma a[3] / thread 2 suma a[2] / thread 5 suma a[5] / thread 1 suma a[1] / thread 4
suma a[4] / thread 0 suma a[0] /
* thread 6 suma= 7
* thread 1 suma= 2
* thread 7 suma= 1
* thread 3 suma= 4
* thread 4 suma= 5
* thread 2 suma= 3
* thread 5 suma= 6
* thread 0 suma= 1
```

```
albduranlopez@albduranlopez:~/Escritorio$ gcc -O2 -fopenmp private-clauseModificado.c -o fuera
albduranlopez@albduranlopez:~/Escritorio$ ./fuera
thread 2 suma a[2] / thread 0 suma a[0] / thread 3 suma a[3] / thread 4 suma a[4] / thread 5 suma
a[5] / thread 1 suma a[1] / thread 6 suma a[6] /
* thread 7 suma= 4196432
* thread 2 suma= 4196434
* thread 5 suma= 4196437
* thread 3 suma= 4196435
* thread 1 suma= 4196433
* thread 0 suma= 8
* thread 6 suma= 4196438
* thread 4 suma= 4196436
```

3. ¿Qué ocurre si en `private-clause.c` se elimina la cláusula `private(suma)`? ¿A qué cree que es debido?

**RESPUESTA:**

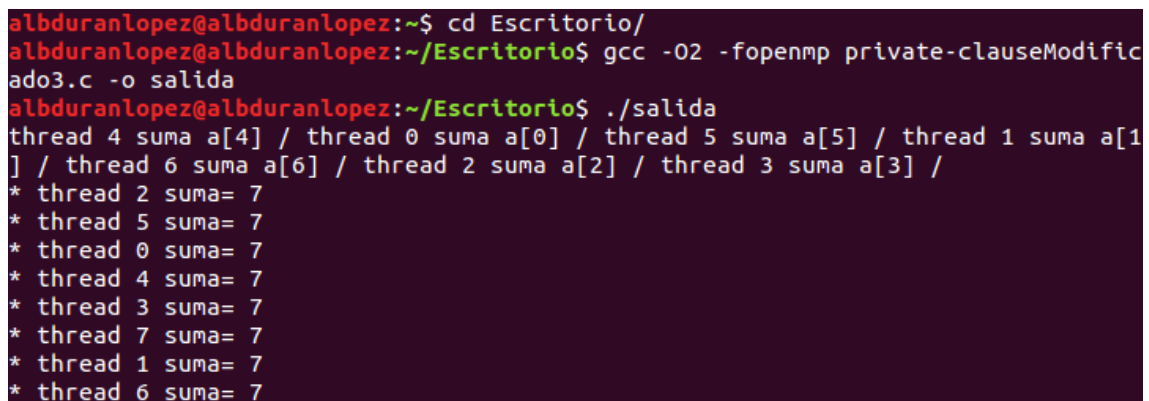
La variable 'suma' pasará a ser compartida por todas las hebras por lo que aparece el mismo resultado en todas ellas, produciendo un resultado incorrecto.

**CÓDIGO FUENTE:** `private-clauseModificado3.c`

```
#include <stdio.h>
#ifdef _OPENMP
#include <omp.h>
#else
#define omp_get_thread_num() 0
#endif
int main()
{
    int i, n = 7;
    int a[n], suma;

    for (i=0; i<n; i++)
        a[i] = i;
    #pragma omp parallel
    {
        suma=1;
        #pragma omp for
        for (i=0; i<n; i++)
        {
            suma = suma + a[i];
            printf("thread %d suma a[%d] / ", omp_get_thread_num(), i);
        }

        printf("\n* thread %d suma= %d", omp_get_thread_num(), suma);
    }
    printf("\n");
}
```

**CAPTURAS DE PANTALLA:**


```
albduranlopez@albduranlopez:~$ cd Escritorio/
albduranlopez@albduranlopez:~/Escritorio$ gcc -O2 -fopenmp private-clauseModificado3.c -o salida
albduranlopez@albduranlopez:~/Escritorio$ ./salida
thread 4 suma a[4] / thread 0 suma a[0] / thread 5 suma a[5] / thread 1 suma a[1]
/ thread 6 suma a[6] / thread 2 suma a[2] / thread 3 suma a[3] /
* thread 2 suma= 7
* thread 5 suma= 7
* thread 0 suma= 7
* thread 4 suma= 7
* thread 3 suma= 7
* thread 7 suma= 7
* thread 1 suma= 7
* thread 6 suma= 7
```

4. En la ejecución de `firstlastprivate.c` de la pag. 21 del seminario se imprime un 6 fuera de la región `parallel`. ¿El código imprime siempre 6 fuera de la región `parallel`? Razone su respuesta.

**RESPUESTA:**

Añadimos: `'lastprivate(suma)'` por lo que nos quedaría así:

```
#pragma omp parallel for firstprivate(suma) lastprivate(suma)
```

SI, imprime siempre 6 porque la última iteración de bucle hace que “suma” valga 6 y `lastprivate(suma)`, que hemos añadido, hace que la última hebra exporte su valor de la variable “suma” a la variable externa

**CAPTURAS DE PANTALLA:**

```
albduranlopez@albduranlopez:~/Escritorio$ gcc -O2 -fopenmp firstlastprivate.c -o firstlast
albduranlopez@albduranlopez:~/Escritorio$ ./firstlast
thread 6 suma a[6] /thread 0 suma a[0] /thread 2 suma a[2] /thread 4 suma a[4] /
thread 1 suma a[1] /thread 5 suma a[5] /thread 3 suma a[3] /
Fuera de la construccion 'parallel for' suma = 6
```

5. ¿Qué ocurre si en `copyprivate-clause.c` se elimina la cláusula `copyprivate(a)` en la directiva `single`? ¿A qué cree que es debido?

**RESPUESTA:**

La hebra que ejecuta el `single` recibe el valor de “a” y `copyprivate(a)` difunde el valor de “a” a otras hebras por lo que al eliminar dicha cláusula, no se produce la difusión anterior y únicamente se inicializan bien aquellas zonas que haya inicializado la hebra que ejecutó el `single`.

**CÓDIGO FUENTE:** `copyprivate-clauseModificado.c`

```
#include <stdio.h>
#include <omp.h>

int main() {
    int n = 9, i, b[n];
    for (i=0; i<n; i++)
        b[i] = -1;

    #pragma omp parallel
    { int a;
      #pragma omp single
      {
          printf("\nIntroduce valor de inicialización a: ");
          scanf("%d", &a );
          printf("\nSingle ejecutada por el thread %d\n",omp_get_thread_num());
      }
      #pragma omp for
      for (i=0; i<n; i++)
          b[i] = a;
    }
    printf("Después de la región parallel:\n");
    for (i=0; i<n; i++) printf("b[%d] = %d\t",i,b[i]);
    printf("\n");
}
```

}

**CAPTURAS DE PANTALLA:**

```
albduranlopez@albduranlopez:~/Escritorio$ gcc -O2 -fopenmp copyprivate-clauseModificado.c -o copyprivate
albduranlopez@albduranlopez:~/Escritorio$ ./copyprivate

Introduce valor de inicialización a: 10

Single ejecutada por el thread 2
Después de la región parallel:
b[0] = 10    b[1] = 10    b[2] = 10    b[3] = 10    b[4] = 10    b[5] = 10
b[6] = 10    b[7] = 10    b[8] = 10
```

6. En el ejemplo `reduction-clause.c` sustituya `suma=0` por `suma=10`. ¿Qué resultado se imprime ahora? Justifique el resultado

**RESPUESTA:** El resultado es el mismo que con `suma=0` pero sumándole 10.

Esto es debido a que se mantiene el valor inicial de la variable 'suma' por lo que al hacer el cambio de variable, el valor inicial de 'suma' es 10.

**CÓDIGO FUENTE:** `reduction-clauseModificado.c`

```
#include <stdio.h>
#include <stdlib.h>

#ifdef _OPENMP
#include <omp.h>
#else
#define omp_get_thread_num() 0
#endif

int main(int argc, char const **argv) {
    int i, n = 20, a[n], suma = 10;

    if(argc < 2){
        fprintf(stderr, "Falta iteraciones\n");
        exit(-1);
    }

    n = atoi(argv[1]);
    if(n > 20){
        n = 20;
        printf("n = %d\n", n);
    }

    for (i = 0; i < n; i++)
```

```

    a[i] = i;
#pragma omp parallel for reduction(+:suma)
for (i = 0; i < n; i++)
    suma += a[i];

printf("Tras 'parallel' suma = %d\n", suma);
}

```

**CAPTURAS DE PANTALLA:**

```

albduranlopez@albduranlopez: ~/Escritorio/Tercera Entrega AC/codigo
albduranlopez@albduranlopez:~/Escritorio/Tercera Entrega AC/codigo$ gcc -O2 -fopenmp reduction-clauseModificado.c -o modificado
albduranlopez@albduranlopez:~/Escritorio/Tercera Entrega AC/codigo$ gcc -O2 -fopenmp reduction-clause.c -o normal
albduranlopez@albduranlopez:~/Escritorio/Tercera Entrega AC/codigo$ ./normal 10
Tras 'parallel' suma = 45
albduranlopez@albduranlopez:~/Escritorio/Tercera Entrega AC/codigo$ ./modificado 10
Tras 'parallel' suma = 55
albduranlopez@albduranlopez:~/Escritorio/Tercera Entrega AC/codigo$ ./normal 330
n = 20
Tras 'parallel' suma = 190
albduranlopez@albduranlopez:~/Escritorio/Tercera Entrega AC/codigo$ ./modificado 330
n = 20
Tras 'parallel' suma = 200
albduranlopez@albduranlopez:~/Escritorio/Tercera Entrega AC/codigo$

```

7. En el ejemplo reduction-clause.c, elimine reduction() de #pragma omp parallel for reduction(+:suma) y haga las modificaciones necesarias para que se siga realizando la suma de los componentes del vector a en paralelo sin usar directivas de trabajo compartido .

**CÓDIGO FUENTE:** reduction-clauseModificado7.c

```

#include <stdio.h>
#include <stdlib.h>

#ifdef _OPENMP
#include <omp.h>
#else
#define omp_get_thread_num() 0
#endif

int main(int argc, char const **argv) {
    int i, n = 20, a[n], suma = 10;

    if(argc < 2){
        fprintf(stderr, "Falta iteraciones\n");
        exit(-1);
    }

    n = atoi(argv[1]);
    if(n > 20){
        n = 20;
        printf("n = %d\n", n);
    }

    for (i = 0; i < n; i++)
        a[i] = i;
#pragma omp parallel
    {
        int suma_local=0;
#pragma omp for
        for (i=0; i<n; i++){
            suma_local += a[i];

```

```

    }
    #pragma omp atomic
    suma += suma_local;
}

printf("Tras 'parallel' suma=%d\n", suma);
}

```

### CAPTURAS DE PANTALLA:

```

albduranlopez@albduranlopez:~/Escritorio/Tercera Entrega AC/codigo$ chmod 777 reduction-clauseModificado7.c
albduranlopez@albduranlopez:~/Escritorio/Tercera Entrega AC/codigo$ gcc -O2 -fopenmp reduction-clauseModificado7.c -o modificado
albduranlopez@albduranlopez:~/Escritorio/Tercera Entrega AC/codigo$ ./modificado 300
n = 20
Tras 'parallel' suma=200
albduranlopez@albduranlopez:~/Escritorio/Tercera Entrega AC/codigo$

```

### Resto de ejercicios

8. Implementar un programa secuencial en C que calcule el producto de una matriz cuadrada, M, por un vector, v1 (implemente una versión para variables globales y otra para variables dinámicas, use una de estas versiones en los siguientes ejercicios):

$$v2 = M \bullet v1; v2(i) = \sum_{k=0}^{N-1} M(i, k) \bullet v(k), i = 0, \dots, N-1$$

NOTAS: (1) el número de filas /columnas N de la matriz deben ser argumentos de entrada al programa; (2) se debe inicializar la matriz y el vector antes del cálculo; (3) se debe asegurar que el programa calcula la suma correctamente imprimiendo todos los componentes del vector resultante, v3, para tamaños pequeños de los vectores (por ejemplo, N = 8 y N=11); (5) se debe imprimir sea cual sea el tamaño de los vectores el tiempo de ejecución del código paralelo que calcula el producto matriz vector y, al menos, el primer y último componente del resultado (esto último evita que las optimizaciones del compilador eliminen el código de la suma).

**CÓDIGO FUENTE:** pmv-secuencial.c

```

// gcc -O2 -fopenmp pmv-secuencia.lc -o producto
#include <stdlib.h>
#include <stdio.h>

#ifdef _OPENMP
    #include <omp.h>
#else
    #define omp_get_thread_num() 0
    #define omp_get_num_threads() 1
#endif

int main(int argc, char** argv)
{
    double time1, time2, diferencia;
    int i, j;

    if (argc<2){
        printf("Introduzca tamaño de la matriz y vector\n");
        exit(-1);
    }

    unsigned int N = atoi(argv[1]); // Máximo N =2^32-1=4294967295
    (sizeof(unsigned int) = 4 B)

    double *vector1 = (double*) malloc(N*sizeof(double));

```

```

double *vector2 = (double*) malloc(N*sizeof(double));
double **datos = (double**) malloc(N*sizeof(double *));

for (i=0; i<N; i++){
    datos[i] = (double*) malloc(N*sizeof(double));
}

//Inicializamos
for (i=0; i<N; i++)
{
    vector1[i] = i;
    vector2[i] = 0;
    for(j=0; j<N; j++)
        datos[i][j] = i+j;
}

time1 = omp_get_wtime();

//Calcular v2
for (i=0; i<N; i++){
    for(j=0; j<N; j++)
        vector2[i] += datos[i][j] * vector1[j];
}

time2 = omp_get_wtime();

//Calculamos la diferencia
diferencia = time2 - time1;

printf("Tiempo(seg.):%f\t / Tamaño:%u\t/ V2[0]=%f V2[%d]=%f\n",
diferencia,N,vector2[0],N-1,vector2[N-1]);

//Imprimimos los componentes si es un tamaño pequeño
if (N<12)
    for (i=0; i<N;i++)
        printf(" V2[%d]=%f\n", i, vector2[i]);

//Liberamos espacio
free(vector1);
free(vector2);
for (i=0; i<N; i++)
    free(datos[i]);
free(datos);

return 0;
}

```



**CAPTURAS DE PANTALLA:**

```

albduranlopez@albduranlopez: ~/Escritorio
albduranlopez@albduranlopez:~/Escritorio$ gcc -O2 -fopenmp pmv-secuencial.c -o producto
albduranlopez@albduranlopez:~/Escritorio$ ./producto 30
Tiempo(seg.):0.000002 / Tamaño:30 / V2[0]=8555.000000 V2[29]=21170.000000
albduranlopez@albduranlopez:~/Escritorio$ ./producto 500
Tiempo(seg.):0.000253 / Tamaño:500 / V2[0]=41541750.000000 V2[499]=103792000.000000
albduranlopez@albduranlopez:~/Escritorio$ ./producto 8
Tiempo(seg.):0.000001 / Tamaño:8 / V2[0]=140.000000 V2[7]=336.000000
V2[0]=140.000000
V2[1]=168.000000
V2[2]=196.000000
V2[3]=224.000000
V2[4]=252.000000
V2[5]=280.000000
V2[6]=308.000000
V2[7]=336.000000
albduranlopez@albduranlopez:~/Escritorio$

```

```

albduranlopez@albduranlopez:~/Escritorio$ gcc -O2 -fopenmp pvm-SecuencialGlobales.c -o globales
albduranlopez@albduranlopez:~/Escritorio$ ./globales 30
Tiempo(seg.):0.000001 / Tamaño:30 / V2[0]=8555.000000 V2[29]=21170.000000
albduranlopez@albduranlopez:~/Escritorio$ ./globales 500
Tiempo(seg.):0.000863 / Tamaño:500 / V2[0]=41541750.000000 V2[499]=103792000.000000
albduranlopez@albduranlopez:~/Escritorio$ ./globales 8
Tiempo(seg.):0.000001 / Tamaño:8 / V2[0]=140.000000 V2[7]=336.000000
V2[0]=140.000000
V2[1]=168.000000
V2[2]=196.000000
V2[3]=224.000000
V2[4]=252.000000
V2[5]=280.000000
V2[6]=308.000000
V2[7]=336.000000
albduranlopez@albduranlopez:~/Escritorio$

```

9. Implementar en paralelo el producto matriz por vector con OpenMP a partir del código escrito en el ejercicio anterior usando la directiva `for`. Debe implementar dos versiones del código (consulte la lección 5/Tema 2):

- una primera que paralelice el bucle que recorre las filas de la matriz y
- una segunda que paralelice el bucle que recorre las columnas.

Use las directivas que estime oportunas y las cláusulas que sean necesarias **excepto la cláusula `reduction`**. Se debe paralelizar también la inicialización de las matrices. Respecto a este ejercicio:

- Anote en su cuaderno de prácticas todos los errores de compilación que se han generado durante la realización del ejercicio y explique cómo los ha resuelto (especifique qué ayudas externas ha usado o recibido).
- Anote todos los errores en tiempo de ejecución que se han generado durante la realización del ejercicio y explique cómo los ha resuelto (especifique qué ayudas externas ha usado o recibido).

NOTAS: (1) el número de filas /columnas  $N$  de la matriz deben ser argumentos de entrada; (2) se debe inicializar la matriz y el vector antes del cálculo; (3) se debe asegurar que el programa calcula la suma correctamente imprimiendo todos los componentes del vector resultante, `v3`, para tamaños pequeños de los vectores (por ejemplo,  $N = 8$  y  $N=11$ ); (5) se debe imprimir sea cual sea el tamaño de los vectores el tiempo de ejecución del código que calcula el producto matriz vector y, al menos,

el primer y último componente del resultado (esto último evita que las optimizaciones del compilador eliminen el código de la suma).

### **CÓDIGO FUENTE : pmv-OpenMP-a.c**

```
// gcc -O2 -fopenmp pmv-OpenMP-a.c -o openMP
#include <stdlib.h>
#include <stdio.h>
#include <omp.h>

#ifdef _OPENMP
    #include <omp.h>
#else
    #define omp_get_thread_num() 0
    #define omp_get_num_threads() 1
#endif

int main(int argc, char** argv)
{
    double time1, time2, diferencia;
    int i, j;

    if(argc<2){
        printf("Introduzca tamaño de la matriz y vector\n");
        exit(-1);
    }

    unsigned int N = atoi(argv[1]); // Máximo N =2^32-1=4294967295
    (sizeof(unsigned int) = 4 B)

    double *vector1 = (double*) malloc(N*sizeof(double));
    double *vector2 = (double*) malloc(N*sizeof(double));
    double **datos = (double**) malloc(N*sizeof(double *));

    for(i=0; i<N; i++){
        datos[i] = (double*) malloc(N*sizeof(double));
    }

    //Inicializamos
    #pragma omp parallel
    {
        #pragma omp for private(j)
        for(i=0; i<N; i++)
        {
            vector1[i] = i;
            vector2[i] = 0;
            for(j=0; j<N; j++)
                datos[i][j] = i+j;
        }
        #pragma omp single
        time1 = omp_get_wtime();

        //Calcular v2
        #pragma omp for private(j)
        for(i=0; i<N; i++){
            for(j=0; j<N; j++)
                vector2[i] += datos[i][j] * vector1[j];
        }

        #pragma omp single
        time2 = omp_get_wtime();
    }
}
```

```

//Calculamos la diferencia
diferencia = time2 - time1;

printf("Tiempo(seg.):%f\t / Tamaño:%u\t/ V2[0]=%f V2[%d]=%f\n",
diferencia,N,vector2[0],N-1,vector2[N-1]);

//Imprimimos los componentes si es un tamaño pequeño
if(N<12)
    for(i=0; i<N;i++)
        printf(" V2[%d]=%f\n", i, vector2[i]);

//liberamos espacio
free(vector1);
free(vector2);
for(i=0; i<N; i++)
    free(datos[i]);
free(datos);

return 0;
}

```

**CÓDIGO FUENTE: pmv-OpenMP-b.c**

```

#include <stdlib.h>
#include <stdio.h>
#include <omp.h>

#ifdef _OPENMP
    #include <omp.h>
#else
    #define omp_get_thread_num() 0
    #define omp_get_num_threads() 1
#endif

int main(int argc, char** argv)
{
    double time1, time2, diferencia;
    int i, j;

    if(argc<2){
        printf("Introduzca tamaño de la matriz y vector\n");
        exit(-1);
    }

    unsigned int N = atoi(argv[1]); // Máximo N =2^32-1=4294967295
    (sizeof(unsigned int) = 4 B)

    double *vector1 = (double*) malloc(N*sizeof(double));
    double *vector2 = (double*) malloc(N*sizeof(double));
    double **datos = (double**) malloc(N*sizeof(double *));

    for(i=0; i<N; i++){
        datos[i] = (double*) malloc(N*sizeof(double));
    }

    //Inicializamos
    for(i=0; i<N; i++)
    {
        vector1[i] = i;
        vector2[i] = 0;
        #pragma omp parallel for
        for(j=0; j<N; j++)
            datos[i][j] = i+j;
    }
}

```

```

}

time1 = omp_get_wtime();

//Calcular v2

for(i=0; i<N; i++)
{
    #pragma omp parallel
    {
        double valor=0;
        #pragma omp for
        for(j=0; j<N; j++)
        {
            valor += datos[i][j] * vector1[j];
        }
        #pragma omp critical
        vector2[i]+=valor;
    }
}

time2 = omp_get_wtime();

//Calculamos la diferencia
diferencia = time2 - time1;

printf("Tiempo(seg.):%f\t / Tamaño:%u\t/ V2[0]=%f V2[%d]=%f\n",
diferencia,N,vector2[0],N-1,vector2[N-1]);

//Imprimimos los componentes si es un tamaño pequeño
if(N<12)
    for(i=0; i<N;i++)
        printf(" V2[%d]=%f\n", i, vector2[i]);

//liberamos espacio
free(vector1);
free(vector2);
for(i=0; i<N; i++)
    free(datos[i]);
free(datos);

return 0;
}

```

**RESPUESTA:** En la versión 'a' no he tenido errores y en la 'b' los he solucionado usando critical en vez de atomic.

**CAPTURAS DE PANTALLA:**

```

albduranlopez@albduranlopez:~/Escritorio$ gcc -O2 -fopenmp pmv-OpenMP-b.c -o openmpB
albduranlopez@albduranlopez:~/Escritorio$ ./openmpB 300
Tiempo(seg.):0.004198 / Tamaño:300 / V2[0]=8955050.000000 V2[299]=22365200.000000
albduranlopez@albduranlopez:~/Escritorio$ ./openmpB 30
Tiempo(seg.):0.000193 / Tamaño:30 / V2[0]=8555.000000 V2[29]=21170.000000
albduranlopez@albduranlopez:~/Escritorio$ ./openmpB 8
Tiempo(seg.):0.001492 / Tamaño:8 / V2[0]=140.000000 V2[7]=336.000000
V2[0]=140.000000
V2[1]=168.000000
V2[2]=196.000000
V2[3]=224.000000
V2[4]=252.000000
V2[5]=280.000000
V2[6]=308.000000
V2[7]=336.000000
albduranlopez@albduranlopez:~/Escritorio$

```

10. A partir de la segunda versión de código paralelo desarrollado en el ejercicio anterior, implementar una versión paralela del producto matriz por vector con OpenMP que use para comunicación/sincronización la cláusula `reduction`. Respecto a este ejercicio:

- Anote en su cuaderno de prácticas todos los errores de compilación que se han generado durante la realización del ejercicio y explique cómo los ha resuelto (especifique qué ayudas externas ha usado o recibido).
- Anote todos los errores en tiempo de ejecución que se han generado durante la realización del ejercicio y explique cómo los ha resuelto (especifique qué ayudas externas ha usado o recibido).

**CÓDIGO FUENTE:** pmv-OpenmMP-reduction.c

```

#include <stdlib.h>
#include <stdio.h>

#ifdef _OPENMP
    #include <omp.h>
#else
    #define omp_get_thread_num() 0
    #define omp_get_num_threads() 1
#endif

int main(int argc, char** argv)
{
    double time1, time2, diferencia;
    int i, j;

    if(argc<2){
        printf("Introduzca tamaño de la matriz y vector\n");
        exit(-1);
    }

    unsigned int N = atoi(argv[1]); // Máximo N =2^32-1=4294967295
    (sizeof(unsigned int) = 4 B)

    double *vector1 = (double*) malloc(N*sizeof(double));
    double *vector2 = (double*) malloc(N*sizeof(double));
    double **datos = (double**) malloc(N*sizeof(double *));

    for(i=0; i<N; i++){
        datos[i] = (double*) malloc(N*sizeof(double));
    }
}

```

```

}

//Inicializamos
for(i=0; i<N; i++)
{
    vector1[i] = i;
    vector2[i] = 0;
    #pragma omp parallel for
    for(j=0; j<N; j++)
        datos[i][j] = i+j;
}

time1 = omp_get_wtime();

//Calcular v2

for(i=0; i<N; i++)
{
    #pragma omp parallel
    {
        double valor=0;
        #pragma omp parallel for reduction(+:valor)
        for(j=0; j<N; j++)
        {
            valor += datos[i][j] * vector1[j];
        }
        vector2[i]+=valor;
    }
}

time2 = omp_get_wtime();

//Calculamos la diferencia
diferencia = time2 - time1;

printf("Tiempo(seg.):%f\t / Tamaño:%u\t/ V2[0]=%f V2[%d]=%f\n",
diferencia,N,vector2[0],N-1,vector2[N-1]);

//Imprimimos los componentes si es un tamaño pequeño
if(N<12)
    for(i=0; i<N;i++)
        printf(" V2[%d]=%f\n", i, vector2[i]);

//liberamos espacio
free(vector1);
free(vector2);
for(i=0; i<N; i++)
    free(datos[i]);
free(datos);

return 0;
}

```

**CAPTURAS DE PANTALLA:**

```

albduranlopez@albduranlopez: ~/Escritorio
albduranlopez@albduranlopez:~/Escritorio$ gcc -O2 -fopenmp pmv-OpenMP-reduction.c -o reduction
albduranlopez@albduranlopez:~/Escritorio$ ./reduction 300
Tiempo(seg.):0.000906 / Tamaño:300 / V2[0]=62685350.000000 V2[299]=89460800.000000
albduranlopez@albduranlopez:~/Escritorio$ ./reduction 30
Tiempo(seg.):0.000536 / Tamaño:30 / V2[0]=51330.000000 V2[29]=105850.000000
albduranlopez@albduranlopez:~/Escritorio$ ./reduction 8
Tiempo(seg.):0.000075 / Tamaño:8 / V2[0]=1120.000000 V2[7]=1344.000000
V2[0]=1120.000000
V2[1]=1008.000000
V2[2]=784.000000
V2[3]=896.000000
V2[4]=756.000000
V2[5]=1120.000000
V2[6]=924.000000
V2[7]=1344.000000
albduranlopez@albduranlopez:~/Escritorio$

```

11. Ayudándose de una hoja de cálculo (recuerde que en las aulas está instalado OpenOffice) realice una tabla y una gráfica que permitan comparar la escalabilidad (ganancia en velocidad en función del número de cores) en atcgrid y en el PC local del mejor código paralelo de los tres implementados en los ejercicios anteriores para dos tamaños (N) distintos (consulte la Lección 6/Tema 2). Usar `-O2` al compilar. Justificar por qué el código escogido es el mejor. NOTA: Nunca ejecute en atcgrid código que imprima todos los componentes del resultado.

**TABLA Y GRÁFICA (por ejemplo para 1-4 threads PC local, y para 1-12 threads en atcgrid, tamaños-N: alguno del orden de cientos de miles):**

**COMENTARIOS SOBRE LOS RESULTADOS:**

Como solo nos piden para dos N cogemos: N=10000 y N=30000.

Para un N mayor atcgrid tarda muchísimo en calcular el tiempo. En realidad, se tarda más en crear las hebras que lo que van a calcular en sí por lo que no he elegido un N mayor.

Como podemos observar en las gráficas, pmv-OpenMP-b.c es el mejor código en atcgrid.

Mientras que en el PC-local prácticamente tardan lo mismo.

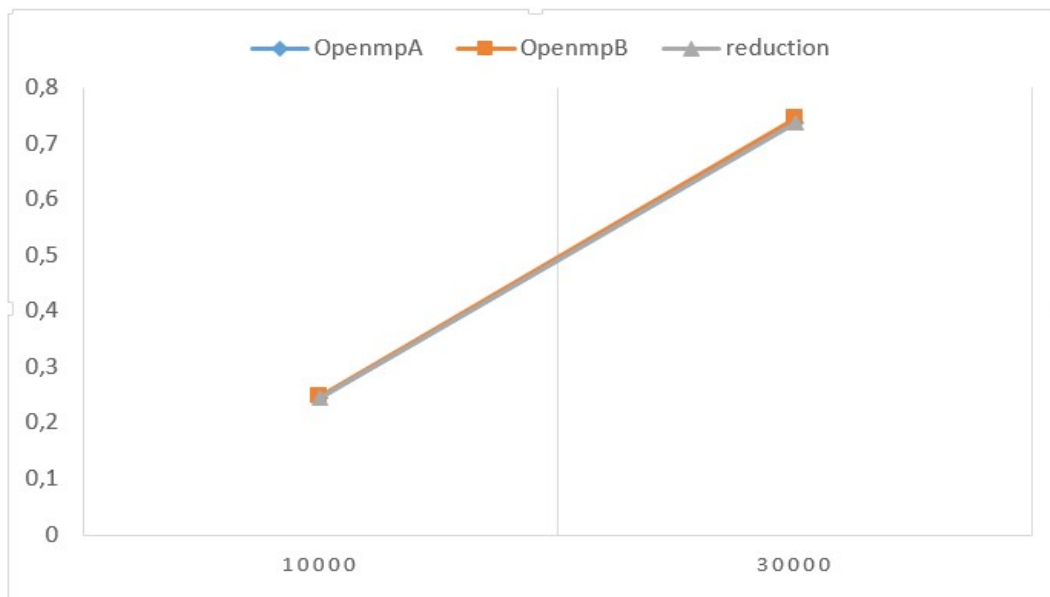
**PC-LOCAL**

N	OpenMP-A	OpenMP-B	Reduction
10000	0,244398	0,249109	0,244321
30000	0,737948	0,747327	0,739596

**ATCGRID**

N	OpenMP-A	OpenMP-B	Reduction
10000	0,134622	0,137684	0,159754
30000	0,413043	0,237986	0,479382

## PC-LOCAL



## ATCGRID

